jamk.fi

# Reverse Engineering TTC6510-3002

## Lab05- 32-bit ELF-binary

AB0168 Mays AL-Azzawi

Information and communication Technology

Degree Programme in Bachelor of Engineering

Jyväskylän ammattikorkeakoulu

JAMK University of Applied Sciences

# 1 Main function of the program

The code begins with the standard preamble. This sets up the stack frame and allocates memory for local variables. The string "serial key: " is printed to prompt the user for input. Input is read using scanf with “%s” as an argument. It is stored in a buffer as a string. This is passed to the check_serial function for validation. The result is checked.

- _time: This function to retrieving the current time.
- _stand: to initialize the serial key number.
- _memset: to initial a block of memory [var_17] set to zero.
- _printf: to print the string “Insert serial key” to the console, prompting the user input.
- ___isoc99_scanf: it is readying user input and check if it is string.
- Check_serial: to check the user input validation.

- word ptr: it means the following operand is a 16-bit value located in memory at a particular address. Reverse assembly is often used in debugging and reverse engineering to understand the low-level details of a program or binary executable.

```
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp        ; sets the value of the ebp register to the current value of the stack pointer (esp).
sub     esp, 48h        ; This instruction subtracts 0x48 (72 in decimal) from the stack pointer, allocating 72 bytes of space on the stack for local variables.
mov     eax, [ebp+argv]
mov     ecx, [ebp+argc]
xor     edx, edx        ; This XORs the edx register with itself, effectively setting it to zero.
mov     [ebp+var_4], 0  ;  This stores the value 0 at the memory address [ebp+var_4]. It initializes a local variable or flag.
mov     dword ptr [esp], 0 ;  This stores the value 0 at the top of the stack. It's common to use the stack for passing arguments to functions.
mov     [ebp+var_1C], eax
mov     [ebp+var_20], ecx
mov     [ebp+var_24], edx
call    _time           ; This is a function call to a function named _time. It's likely retrieving the current time
mov     [esp], eax
call    _srand          ; This is a function call to a function named _srand. It's likely initializing the random number generator.
xor     eax, eax
lea     ecx, [ebp+User_input]
mov     [esp], ecx
mov     dword ptr [esp+4], 0
mov     dword ptr [esp+8], 13h ; This stores the value 0x13 (19 in decimal) eight bytes above the top of the stack.
mov     [ebp+var_28], eax
call    _memset         ; This is a function call to _memset, which is likely used to initialize a block of memory. In this case, it's initializing var_17 with zeros.
lea     eax, aInsertSerialKe ; "Insert serial key: "
mov     [esp], eax      ; This stores the memory address of the string (in eax) at the top of the stack. It prepares the string to be printed by printf.
call    _printf         ; : This is a function call to _printf to print the string "Insert serial key: " to the console, prompting the user for input.
lea     ecx, [ebp+User_input]
lea     edx, a%s        ; This loads the effective address of the string "%s" into edx. This string is often used as a format specifier for input functions like scanf, indicating that a string should be read from the user
mov     [esp], edx
mov     [esp+4], ecx
mov     [ebp+var_2C], eax
call    ___isoc99_scanf ;  This is a function call to ___isoc99_scanf, which is a standard C library function for reading input from the user based on the format specifier "%s." It will read a string from the user and sto
lea     ecx, [ebp+User_input] ; This loads the effective address of the local variable [ebp+var_17] (which should now contain the user's input) into ecx.
mov     [esp], ecx
mov     [ebp+var_30], eax
call    check_serial    ; This is a function call to check_serial, which is likely responsible for verifying whether the user's input (serial key) is valid.
and     al, 1           ;  This bitwise AND operation with al (the lowest byte of the eax register) and 1 is used to check if the least significant bit of al is set. It's a way to extract a boolean value from al
mov     [ebp+var_18], al
test    [ebp+var_18], 1 ; If it is zero (false), the zero flag (ZF) will be set.
jz      loc_8049660     ; : This is a conditional jump instruction. It jumps to the address loc_8049660 if the zero flag is set, indicating that the result of the check was false or zero
```

Figure 1 main function

```
lea     eax, aSerialOkStarti ; "serial ok, starting game!\n"
mov     [esp], eax
call    _printf
mov     [ebp+var_34], eax
call    start_game
jmp     loc_804966E     ; This instruction XORs the eax register with itself, effectively setting it to zero. This is a common way to clear a register or variable.

loc_8049660:
lea     eax, aBadSerialKeyEx ; "bad serial key, exiting...!\n"
mov     [esp], eax
call    _printf

c_804966E:              ; This instruction XORs the eax register with itself, effectively setting it to zero. This is a common way to clear a register or variable.
r       eax, eax
d       esp, 48h        ; This instruction adds 0x48 (72 in decimal) to the stack pointer (esp). It's undoing the stack space allocation that was done at the beginning of the function, essentially deallocating the local variables and any other stack space used by the func
p       ebp
tn
in endp
```

Figure 2 main function arguments

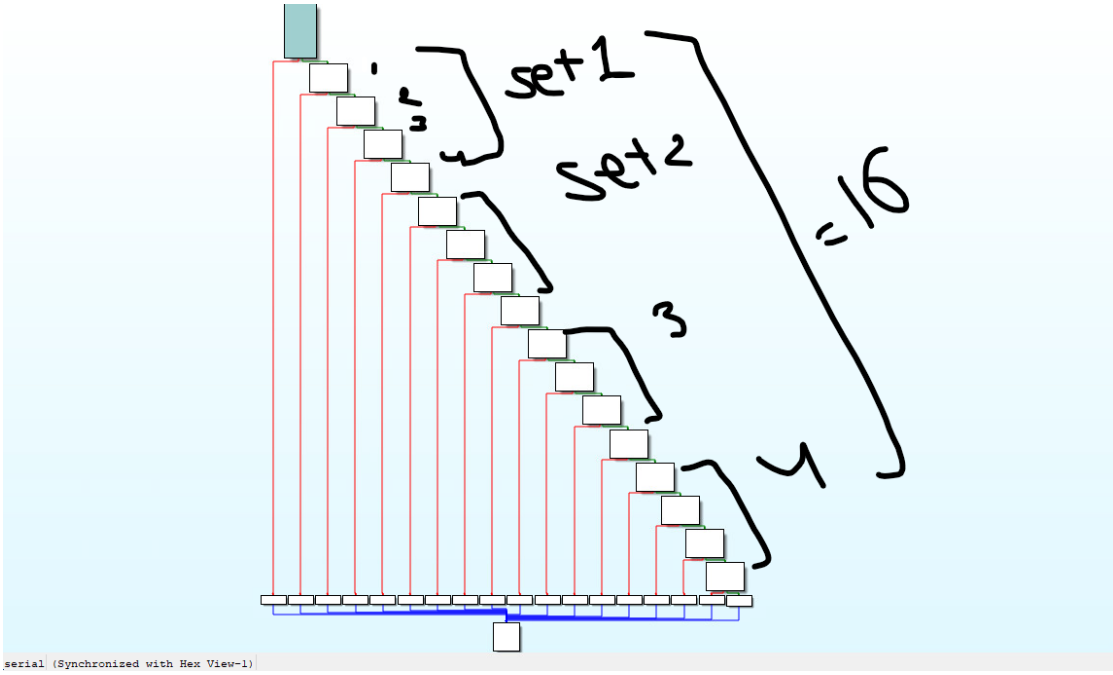
## 1.1 Check serial function

The begin of this function start with set of codes to check the length of a string as user input, it is obtained from the function argument =19 characters, and if True jump to location in the memory call loc_804920A for further instructions.

```
; Attributes: bp-based frame

public check_serial
check_serial proc near

var_8= dword ptr -8
var_1= byte ptr -1
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 18h        ;  This subtracts 0x18 (24 in decimal) from the stack pointer, allocating 24 bytes of space on the stack for local variables
mov     eax, [ebp+arg_0] ; This loads the value at the memory address [ebp+arg_0] into the eax register. It seems to be fetching the value of a function argument
mov     ecx, [ebp+arg_0]
mov     edx, esp
mov     [edx], ecx
mov     [ebp+var_8], eax
call    _strlen         ; his is a function call to _strlen, which is likely a standard C library function for calculating the length of a null-terminated string. The result is likely stored in the eax register
cmp     eax, 13h        ; his is a function call to _strlen, which is likely a standard C library function for calculating the length of a null-terminated string. The result is likely stored in the eax register
jz      loc_804920A     ; This is a conditional jump instruction. It jumps to the address loc_804920A if the previous comparison (cmp) resulted in equal values. In other words, it checks if the length of the string is 19 characters and jumps accordingly
```

After checking the condition, the function starts to set the serial instructions and from the construction of the flow of the code we found:

- The key has 4 sets and in total are 16 and as we know that the key should be 19 so still 3 characters are still missing.




```
loc_804920A:
call    ___ctype_b_loc
mov     eax, [eax]
mov     ecx, [ebp+arg_0]
movsx   ecx, byte ptr [ecx] ; this instruction loads an 8-bit (byte-sized) value from memory at the
                            ;  address calculated by adding 1 to the value in the ecx register.
                            ;  The loaded byte is then sign-extended to a 32-bit value and placed
                            ;  into the ecx register. This is often used when working with signed
                            ;  values to ensure that the sign bit is correctly extended to fill the
                            ;
                            ;  larger register.
movzx   eax, word ptr [eax+ecx*2] ;  this instruction loads a 16-bit (word-sized)
                            ;  value from memory at the address calculated by adding ecx multiplied
                            ;  by 2 to the value in the eax register. The value is then zero-extended
                            ;  and placed into the eax register
and     eax, 800h
cmp     eax, 0
jnz     loc_8049232
```

- Notice the code it is increasing by one and every 4 steps will pass one, so from the code below we can see the first 4 codes it is loading 8-bits(byte) from the memory and add 1 to the value in the register ecx and on the step 5 it is pass the 5th character so we know the forth character is missing and this is continue for the other 3 sets so, we can set it to – or / or any other special character except white space as we tested on the console, Now we know we have key set like xxx-xxxx-xxx-xxx which in total it is 19

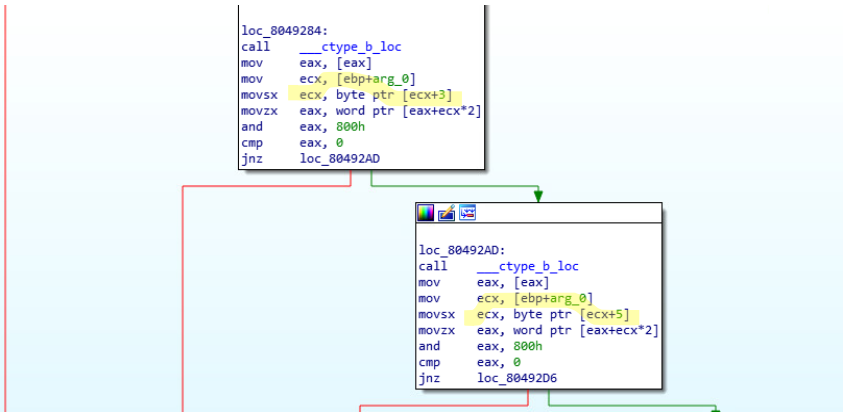
movsx ecx, byte ptr [ecx]

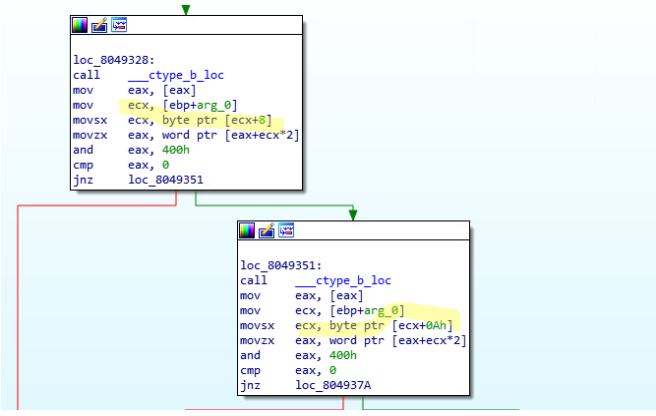
movsx ecx, byte ptr [ecx+1]

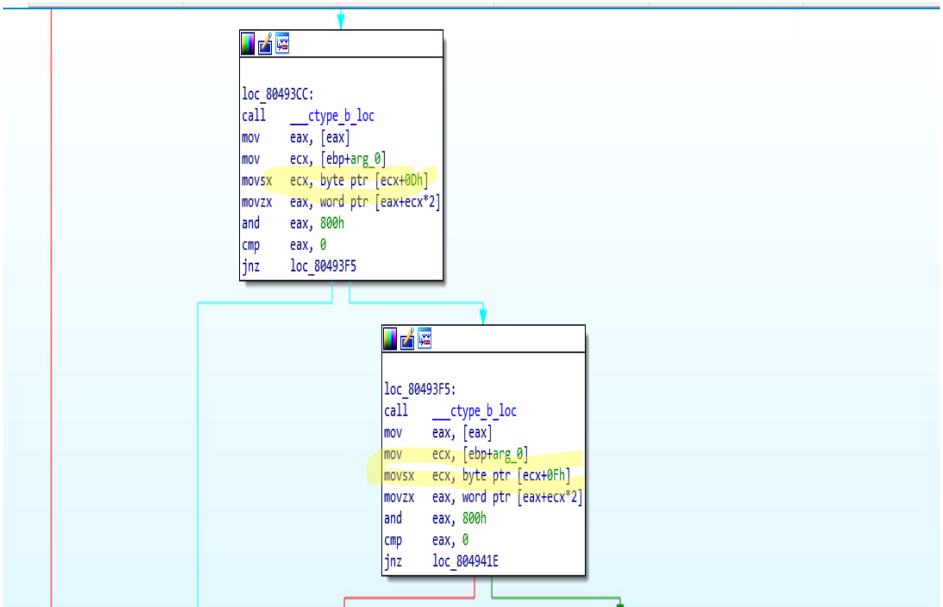
movsx ecx, byte ptr [ecx+2]

movsx ecx, byte ptr [ecx+3]

movsx ecx, byte ptr [ecx+5]



- The second step was to find that there are only two hexa numbers used as un argument with ‘and’ which are 400h and 800h

  400h=0000 0100 0000 0000

  800h=0000 1000 000 0000

```
loc_80493A3:
call    ___ctype_b_loc
mov     eax, [eax]
mov     ecx, [ebp+arg_0]
movsx   ecx, byte ptr [ecx+0Ch]
movzx   eax, word ptr [eax+ecx*2]
and     eax, 800h       ; he and operation compares each bit position between the two operands
                        ;  and results in a new value where each bit is set to 1 only if both
                        ;  corresponding bits in eax and 800h are 1. In this case, since only the
                        ;  leftmost bit in 800h is set to 1, all other bits in the result will be
                        ;  0:
cmp     eax, 0
jnz     loc_80493CC
```

From this point I started to digging to know what the actual job of the function is __ctype_b_loc and I found that this function:

- The __ctype_b_loc() function shall return a pointer into an array of characters in the current locale that contains characteristics for each character in the current character set.

Table 1 function's arrangement

| Bit number | Value | Description |
|---|---|---|
| 7 | 1 | graphical |
| 6 | 1 | printing |
| 5 | 0 | whitespace |
| 4 | 1 | hexadecimal numeric |
| 3 | 0 | numeric |
| 2 | 1 | alphabetic |
| 1 | 0 | lowercase |
| 0 | 1 | uppercase |
| | 0 | |
| | 0 | |
| | 0 | |
| | 0 | |
| 11 | 1 | alphanumeric |
| 10 | 0 | punctuation |
| 9 | 0 | control |
| 8 | 0 | blank |

Now to apply this on the two hexa number we had in our program, we found

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 800 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 400 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | number | alpha | | | | | | | | | | |

Table 2 arrangement for  hexa numbers base on _ctype_b_loc function

it is return number on 800h and alphabets on 400h.

In this point we still missing the arrange of the set of the key and from our code, I followed the use of 400h and 800h found it is return a set like this:

NAAN-NNAA-AANN-NNNA

*’N ‘presents 800h which is a Number and ‘A ‘presents 400h which is a letter.

# 2 The Result

The serial key is 4 sets divided by -, it is alpha and numbers with no white space. The character set base on the flow of the code as show in the figure above and tested on

5 different keys fellow the same instruction to generate a key base on the code instruction.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 800 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 400 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | number | alpha | | | | | | | | | | |

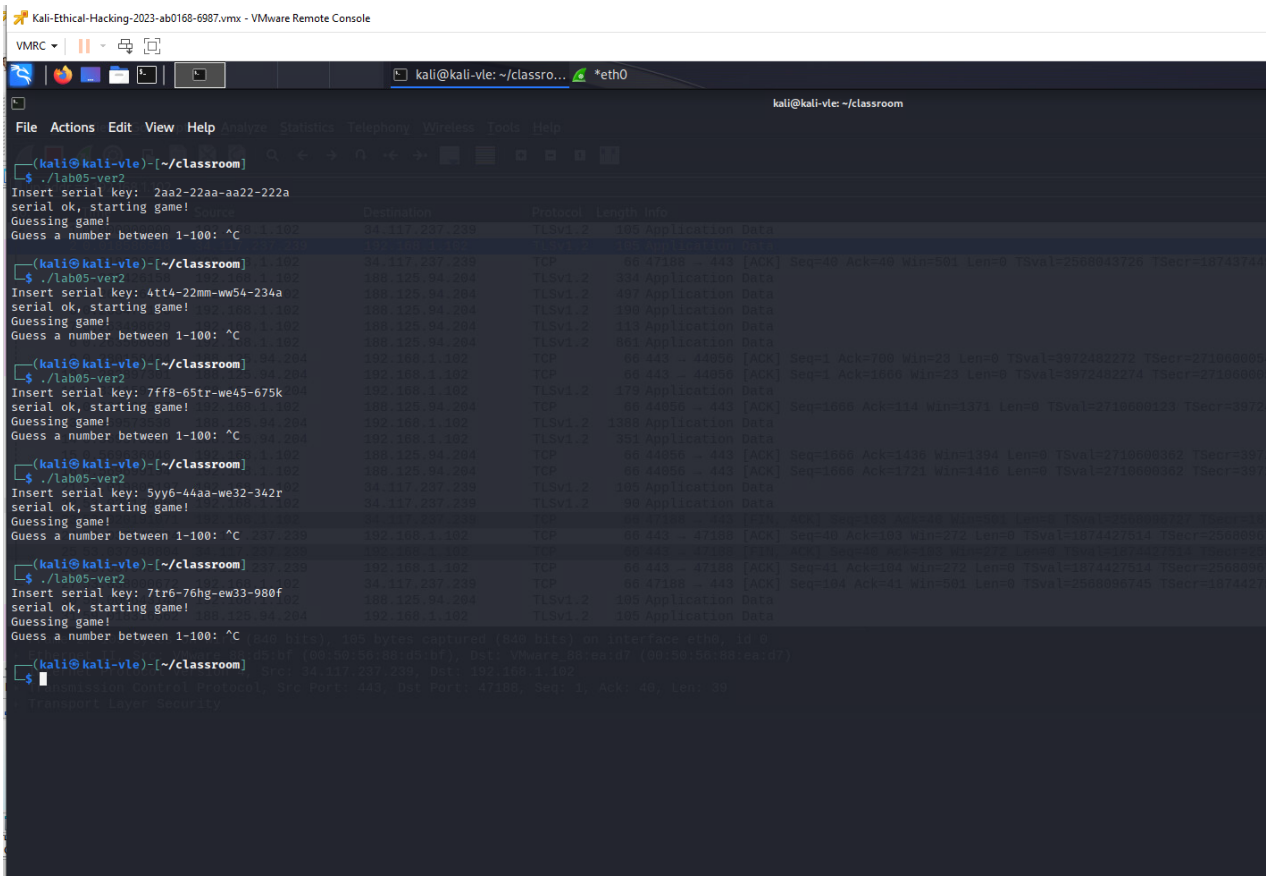
NAAN-NNAA-AANN-NNNA= 2aa2-22aa-aa22-222a

4tt4-22mm-ww54-234a

7ff8-65tr-we45-675k

5yy6-44aa-we32-342r

7tr6-76hg-ew33-980f




# 3 References


- https://braincoke.fr/blog/2018/05/what-is-ctype-b-loc/

# 4 Tracking information for lab05

| Topic | Date | Time Used | Description |
|---|---|---|---|
| Begin Lab05 and Static analysis | 29.09.2023 | 7h | Study the material and Search result and try to execute the lab |
| Continue lab-05 | 30.09.2023 | 20h | Figure out the functionality of the lab |
| Report | 1.10.2023 | 3h | Documenting, writing down the finding |
| Total | | 30h | Success |